PaSCAni: lenguaje para validación y verificación de requerimientos funcionales en tiempo de ejecución

Presentado por:

Fabián Andrés Caicedo Cuellar
Miguel Ángel Jiménez Achinte

Directores:

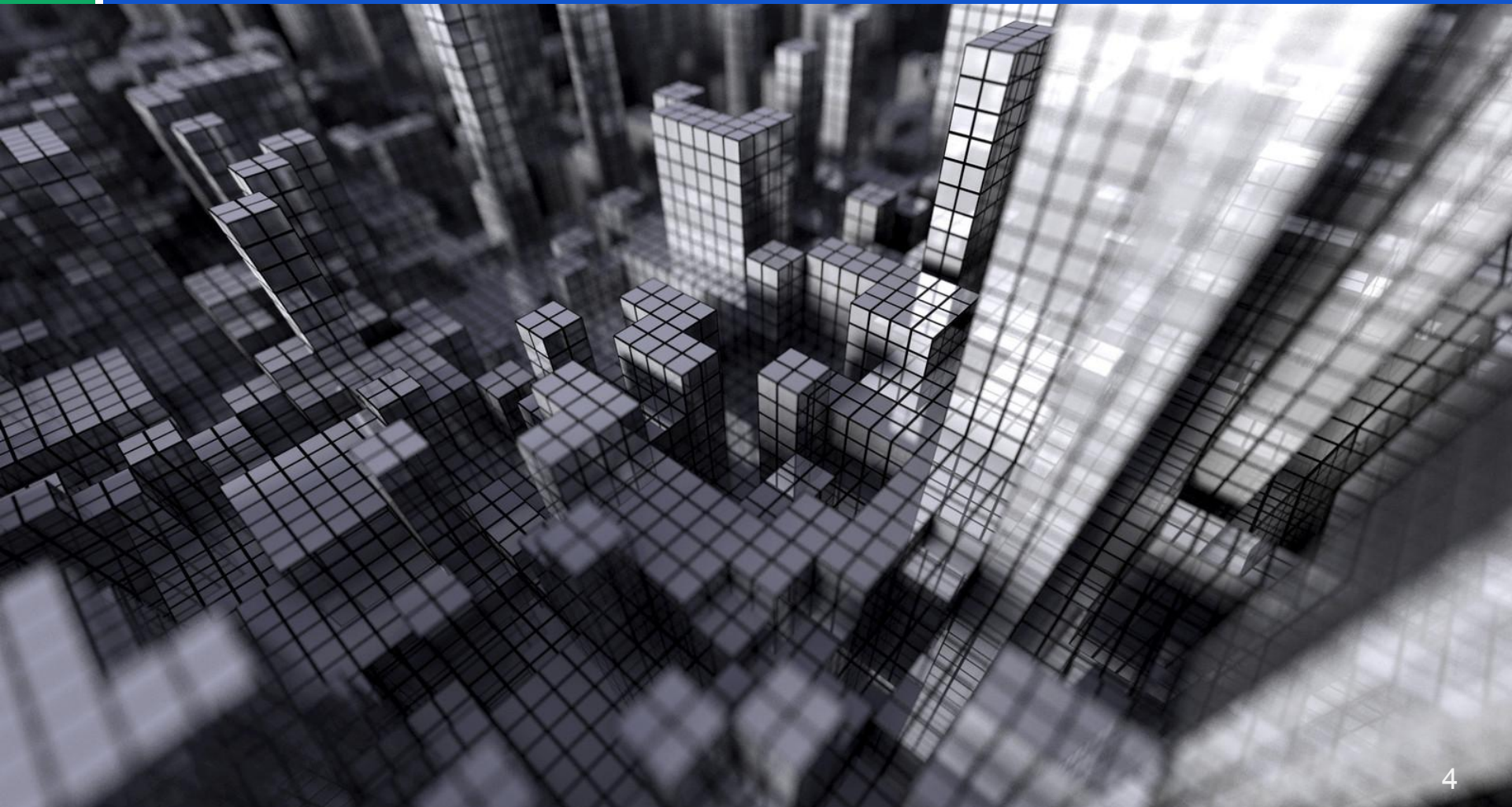
Gabriel Tamura Morimitsu
Ángela Patricia Villota Gómez

Contenido

1. Introducción
2. Planteamiento del problema
3. Objetivos
4. Solución y contribución
5. Metodología
6. Resultados y Conclusiones
7. Trabajo futuro

Contenido

Escala: Sistemas tradicionales y ULSS

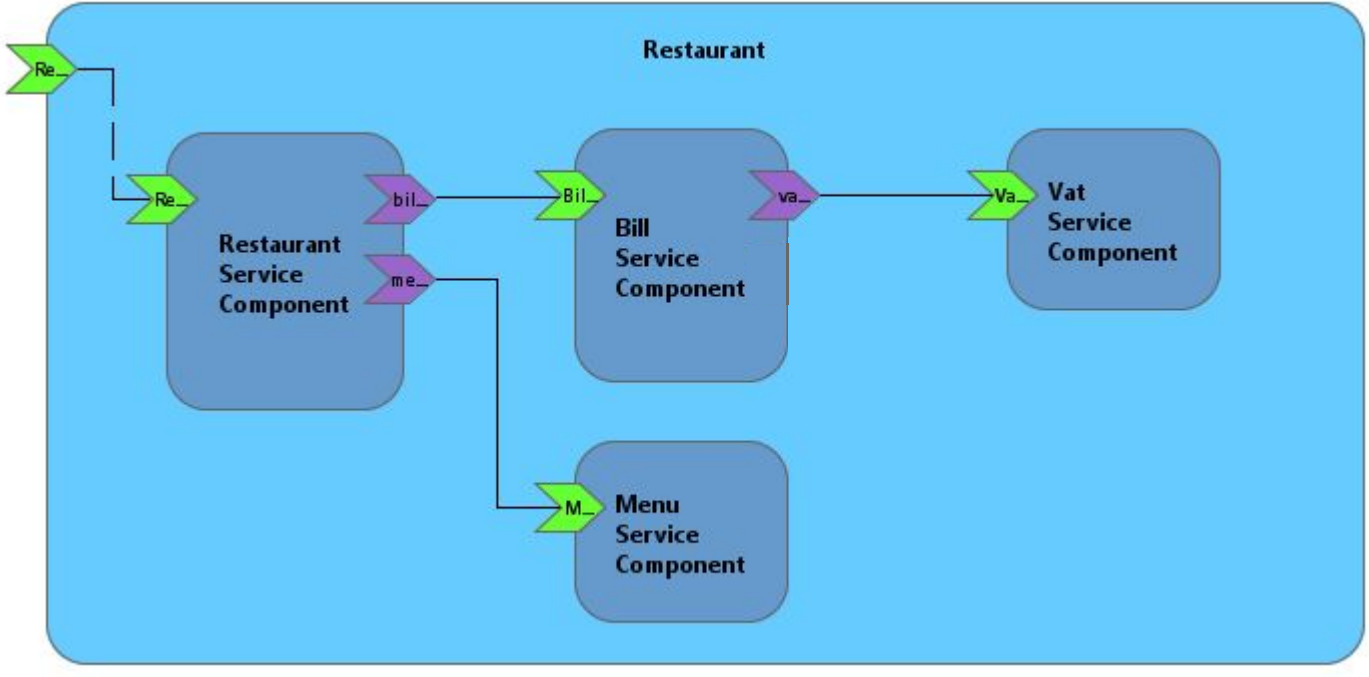


Sistemas a Gran Escala (ULSS)

- Control descentralizado.
- Evolución y despliegue continuo.
- Las fallas son comunes.
- Elementos heterogéneos, inconsistentes y cambiantes.
- Requerimientos inherentemente contradictorios, desconocidos y diversos.

Sistemas de Software Auto-adaptativos (SAS)

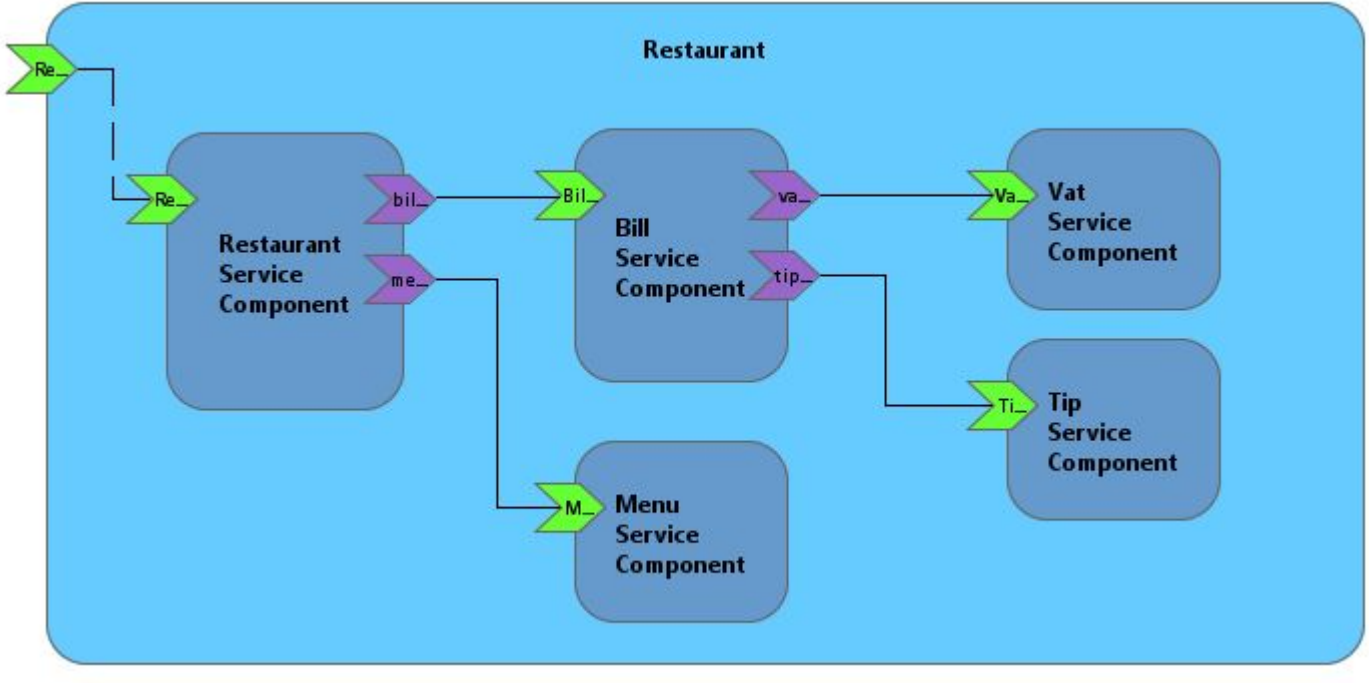
- Reconfiguración estructural de la arquitectura.



- **RestaurantServiceComponent**
- **MenuServiceComponent**
- **BillServiceComponent**
- **VATServiceComponent**

Sistemas de Software Auto-adaptativos (SAS)

- Reconfiguración estructural de la arquitectura.



- RestaurantServiceComponent
- MenuServiceComponent
- BillServiceComponent
- VATServiceComponent
- TipServiceComponent

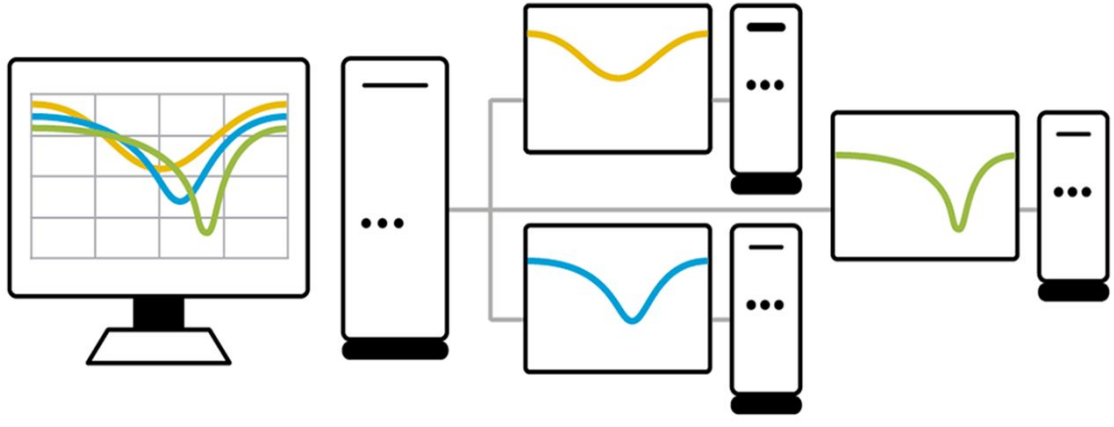
ULSS y SAS: Internet

Necesidad de la auto-adaptación: cumplimiento de los objetivos de negocio.

Sistemas de software basados en componentes (1)

¿Qué es una aplicación?

Una forma de pensarlo es como un conjunto de componentes de software que trabajan juntos.



Sistemas de software basados en componentes (2)

Independientemente de cual sea la organización de una aplicación, se requieren dos cosas:

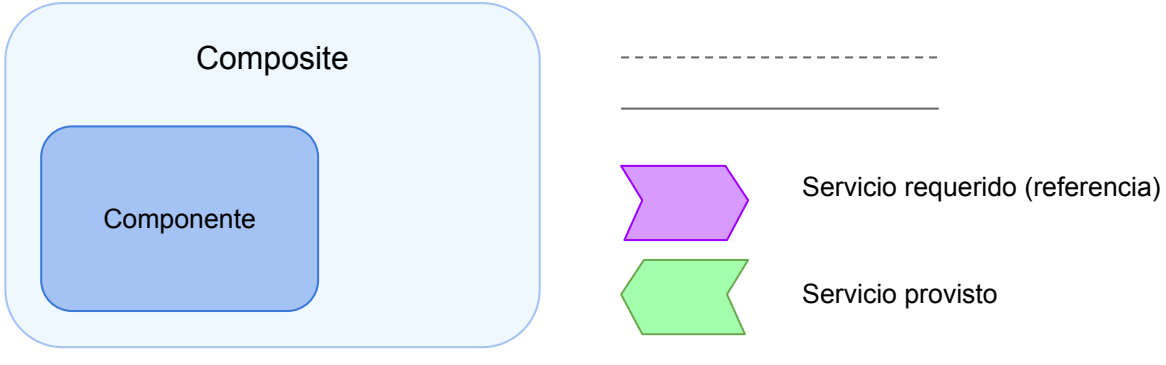
- Una manera de crear componentes de software.
- Y un mecanismo para describir cómo estos componentes trabajan juntos.

D. Chappel. Introducing SCA. Technical report, Chappell & Associates, 2007.

Estándar SCA: Arquitectura de componentes basada en servicios

- SCA define una propuesta general para hacer ambas cosas (creación de componentes y definición de un mecanismo para describir la comunicación entre estos).
- Adicionalmente, los componentes descritos en SCA pueden estar implementados en diferentes tecnologías (Java, BPEL, WSDL, RESTful, entre otros).

En SCA, esto se representa usando:



FraSCAti

OW2 FraSCAti es un framework de componentes que implementa la especificación de SCA.

- Permite reconfiguración de la arquitectura del software en tiempo de ejecución.
- FraSCAti Explorer

Motivación

- Actualmente, las aplicaciones de software enfrentan cambios de contexto no predecibles, incluyendo cambios en los objetivos de negocio de las organizaciones.

- Los sistemas de software deben **satisfacer los requerimientos que cambian** en **tiempo de ejecución**.

- Aunque altos niveles de adaptación y autonomía proveen nuevas formas de operación para los sistemas de software en entornos altamente dinámicos, el **desarrollo de métodos de V&V** para garantizar el cumplimiento de los objetivos de los sistemas auto-adaptativos es uno de los principales retos que enfrenta este campo de investigación.

- CASTAÑEDA BUENO, Lorena. A Reference Architecture for Component-Based Self-Adaptive Software Systems. Santiago de Cali, 2012, Tesis Maestría (Máster en Gestión de Informática y Telecomunicaciones concentración en Ingeniería del Software). Universidad Icesi. Facultad de Ingeniería.
- G. Tamura, N. M. Villegas, H. Muller, J. P. Sousa, B. Becker, G. Karsai, S. Mankovskii, M. Pezze, W. Schafer, L. Tahvildari, and K. Wong. Towards practical runtime verification and validation of self-adaptive software systems. Software Engineering for Self-Adaptive Systems II, Springer, 2012.

Nuevos enfoques y retos desde los SAS y ULSS

- Los sistemas ULS requerirán lenguajes que permitan expresar una visión de abstracción más amplia.

  [R1] Diseñar un lenguaje de programación que permita:

  - Hacer una abstracción sobre un conjunto de componentes del sistema
  - Establecer *restricciones* sobre los requerimientos funcionales
  - Facilitar la V&V para configuraciones diferentes de los servicios y los componentes.

- Los lenguajes actuales separan las preocupaciones de manera pobre, representan información del dominio de la solución en lugar del dominio del problema.

P. Feiler, R. P. Gabriel, J. Goodenough, R. Linger, T. Longstaff, R. Kazman, M. Klein, L. Northrop, D. Schmidt, K. Sullivan, and K. Wallnau. Ultra-large-scale systems: The software challenge of the future. Technical report, The Carnegie Mellon Software Engineering Institute, 2006.

Nuevos enfoques y retos desde los SAS y ULSS

- Los sistemas de software auto-adaptativos introducen una serie de retos sobre las fases del ciclo de vida, y en general en la ingeniería de software.

  [R2] Proveer evidencia de que un conjunto de propiedades establecidas (o emergentes) están siendo satisfechas **durante la operación del sistema**.

  - Implementar un componente de software que permita realizar V&V en tiempo de ejecución.

Carencia de semántica en los lenguajes de programación

Decir lo que debes → Decir lo que *quieres.*

— Alex russell

¿Es la semántica que tenemos, la semántica que queremos?

— Bruce Lawson

Contenido

Planteamiento del problema (1)

- Adaptación del software mediante la ejecución de tareas de reconfiguración estructural en la arquitectura.

- Esto sugiere la existencia de mecanismos de pruebas no sólo durante el período de producción del software, sino también en un ambiente de operación

Contenido

Objetivo general

Desarrollar e implementar un prototipo funcional de validación y verificación en tiempo de ejecución, capaz de verificar que los cambios estructurales en la arquitectura de un sistema de software auto-adaptativo objetivo cumplen con un conjunto de requerimientos funcionales especificados.

Objetivos específicos

- Proponer un conjunto de reglas léxicas, sintácticas y semánticas que permitan especificar requerimientos funcionales para generar y ejecutar clases de prueba, en tiempo de ejecución.

- Implementar un prototipo funcional de un componente de software que permita realizar validación y verificación de un sistema auto-adaptativo en tiempo de ejecución, empleando un conjunto de especificaciones usando las reglas léxicas, sintácticas y semánticas del lenguaje propuesto.

- Integrar el prototipo funcional propuesto al caso de estudio *A variability model for generating SCA-based matrix-chain multiplication software products.*

Contenido

Propuesta de solución

La propuesta consiste en el diseño y la implementación de un lenguaje de dominio específico.

A partir de esto surgen los siguientes interrogantes:

- ¿Por qué un lenguaje?
- ¿Por qué no usar uno ya existente o extenderlo?

Para el desarrollo de la propuesta

El desarrollo está conformado por 3 partes:

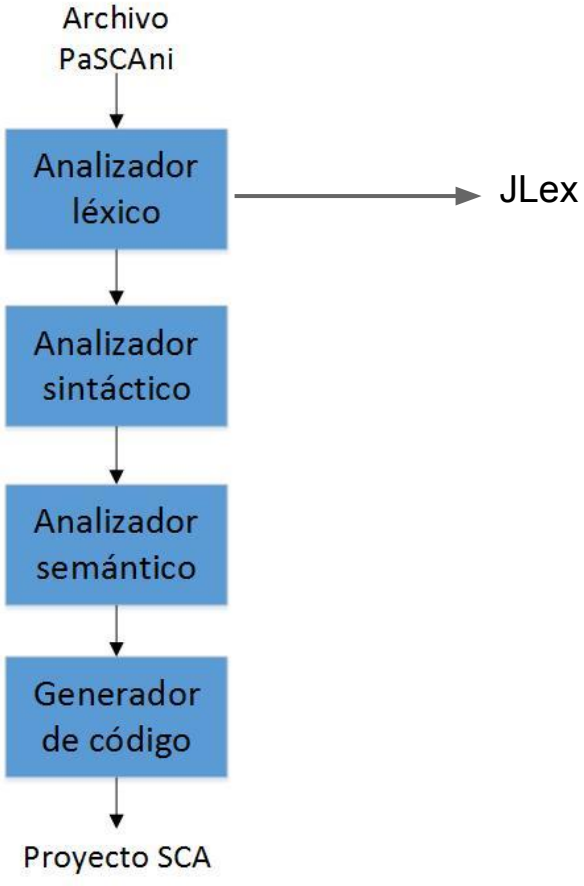
1. Diseño de la gramática independiente de contexto

2. Implementación de un Compilador

    a. Analizador léxico
    b. Analizador sintáctico
    c. Analizador semántico
    d. Generador de código

3. Implementación del modelo de ejecución

Analizador léxico



Analizador léxico: recibe como entrada el archivo PaSCAni y produce una salida compuesta de tokens

JLex: es un generador de analizadores léxicos

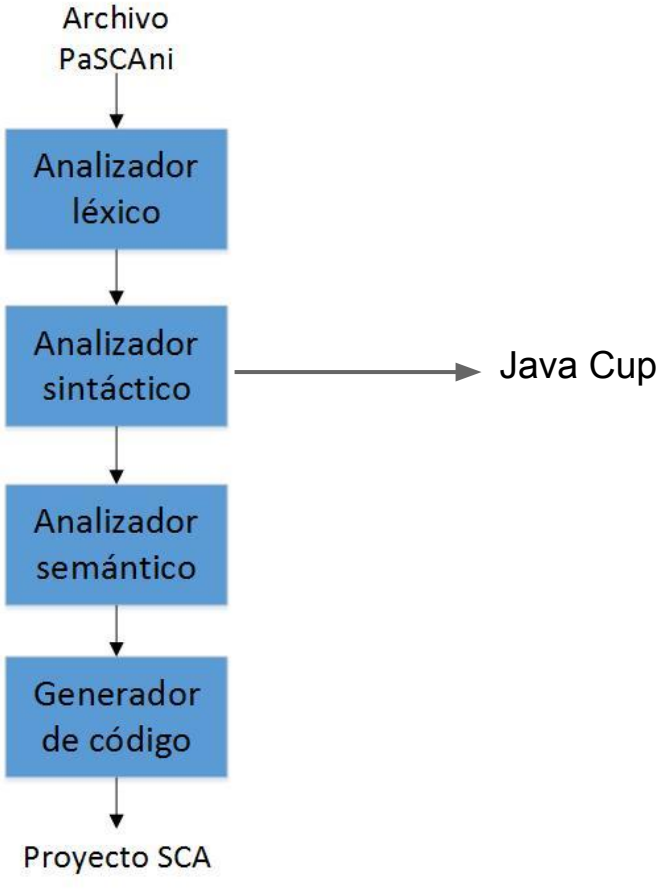
Permite definir las palabras reservadas del lenguaje como:

```
[A-Za-z_][A-Za-z_0-9]*
{return new Symbol(sym.IDENTIFIER, yyline + 1, yychar, yytext());}
```

Mediante el uso de directivas de JLex modificamos la salida:

```
%unicode %cup %char %line
```

Analizador sintáctico



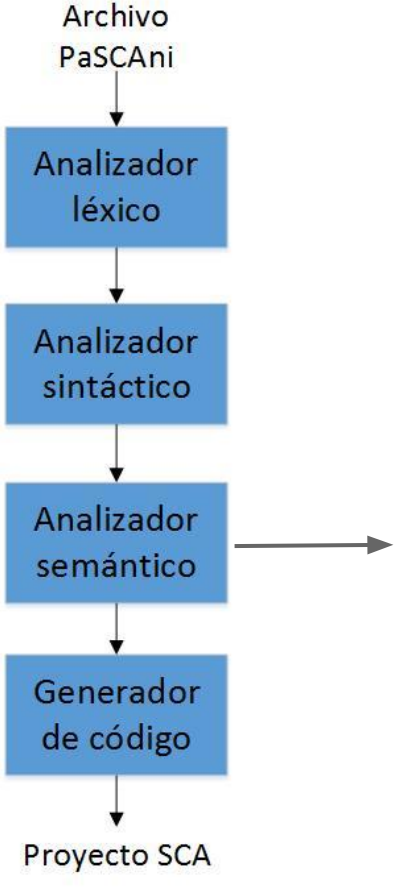
Analizador sintáctico: recibe como entrada en este caso la salida del analizador léxico y crea un árbol de sintaxis abstracto

Java Cup: es un generador de analizador sintáctico, que a partir de la especificación de una gramática, su salida es un parser escrito en java.

Ejemplo de regla de producción:

```
import_statement
     ::= IMPORT qualified_name SEMICOLON
         | JAVA_IMPORT qualified_name DOT TIMES
SEMICOLON
         | JAVA_IMPORT qualified_name SEMICOLON
```

Analizador semántico



La estructura está, sintácticamente, bien escrita. Sin embargo, en el dominio del lenguaje, ¿tiene algún significado?

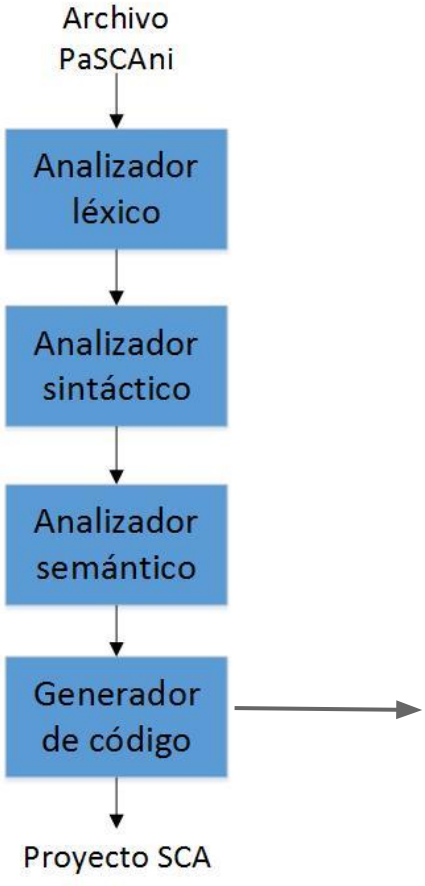
- Chequeo de tipos de datos
- Existencia y alcance de las variables
- Pertinencia de tipos de datos al usar operadores aritméticos, lógicos y relacionales
- Uso de la tabla de símbolos

```
boolean var1 = true
int var2 = 2
```

```
variable_declaration
::= type IDENTIFIER EQUAL_SIGN IDENTIFIER ARITMETIC_OP
IDENTIFIER
```

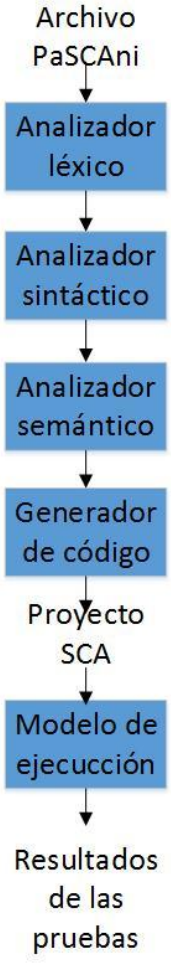
```
int suma = var1 + var2
```

Generador de código



- Los componentes de software a probar están escritos bajo el estándar SCA. Por lo tanto, los componentes de prueba también.
- Los módulos de prueba contienen la información básica para la generación del código.
- La generación de archivos se realizó por medio de plantillas de código (Google Closure Templates)

Modelo de ejecución



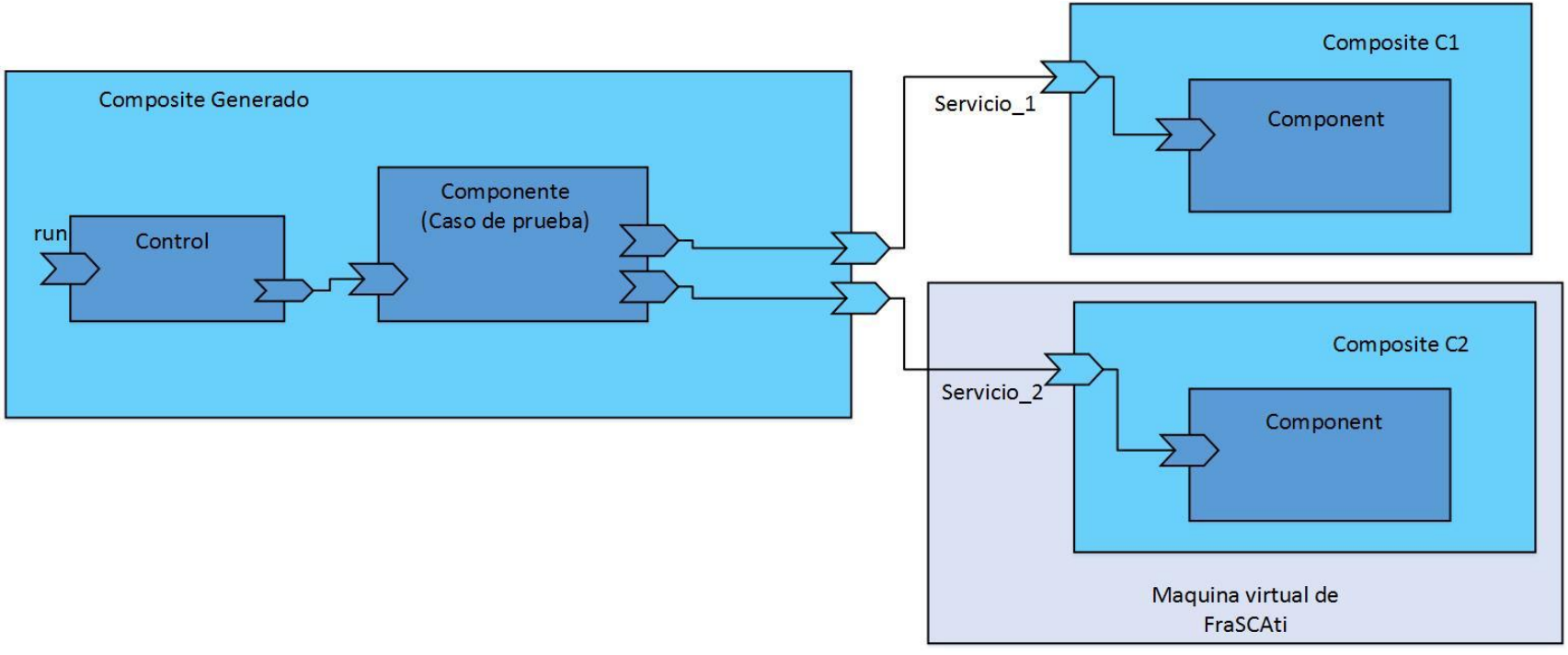
El modelo de ejecución se encarga de hacer el despliegue de los componentes.

En el diseño de este modelo de ejecución se implementaron 2 soluciones:

i). Exigir al programador que los `composites` a probar ya están corriendo en un proceso de FraSCAti.

ii). Realizar su ejecución de manera automática desde PaSCAni.

Modelo de ejecución

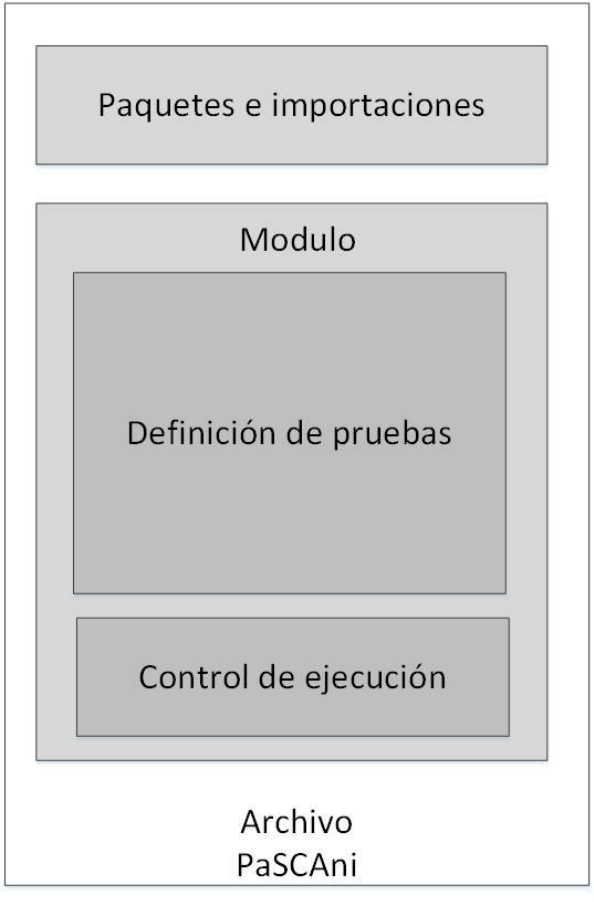


El modelo de ejecución se encarga de desplegar los componentes que no se encuentran actualmente en ejecución en la máquina virtual de FraSCAti.

Estructura de un módulo de prueba en PaSCAni

Un archivo PaSCAni tiene la siguiente estructura:

- Definición de paquete y conjunto de archivos importados.
- Definición del módulo de prueba
  - Definición de las pruebas
    - Definición de componentes
    - Definición de casos de prueba
  - Definición de control de ejecución.



Estructura de un módulo de prueba en PaSCAni

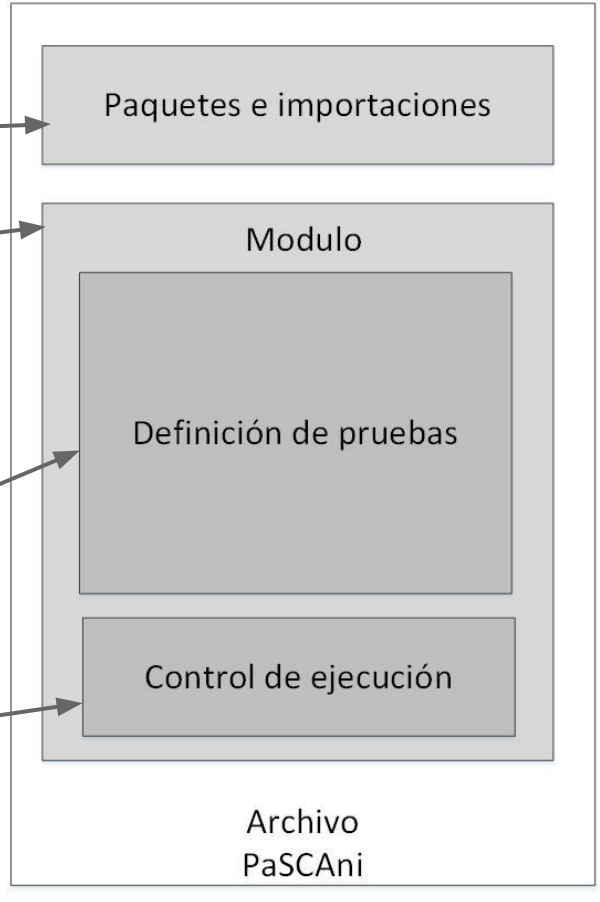
```
package org.driso.prueba;
java-import javaclass;
import moduloUno;

modulo moduloPrueba {

    composite componente = ....

    testsuite casoUno(...) using compo.servi {
    }
    testsuite casoDos(...) using compo.servi {
    }

    control {
    }
}
```



Estructura de un módulo de prueba en PaSCAni

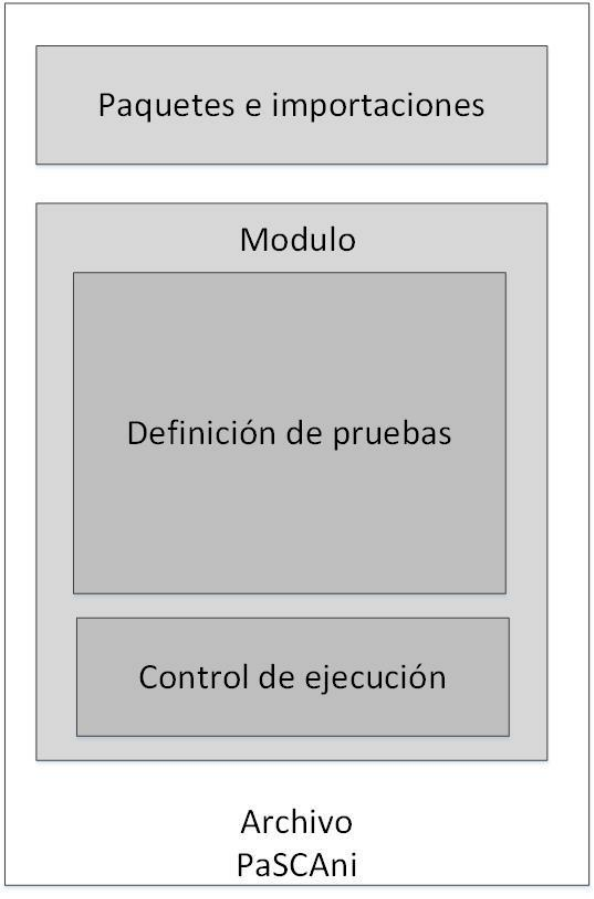
La definición de pruebas está compuesta de dos partes:

- Definición de los componentes

```
composite comp = loadComposite( ... ) providing services { ...
                                    }
                  using libraries { ... }
```

- Definición de los casos de prueba

```
testsuite casoDePrueba ( ... )  {
     return ...
 }
```



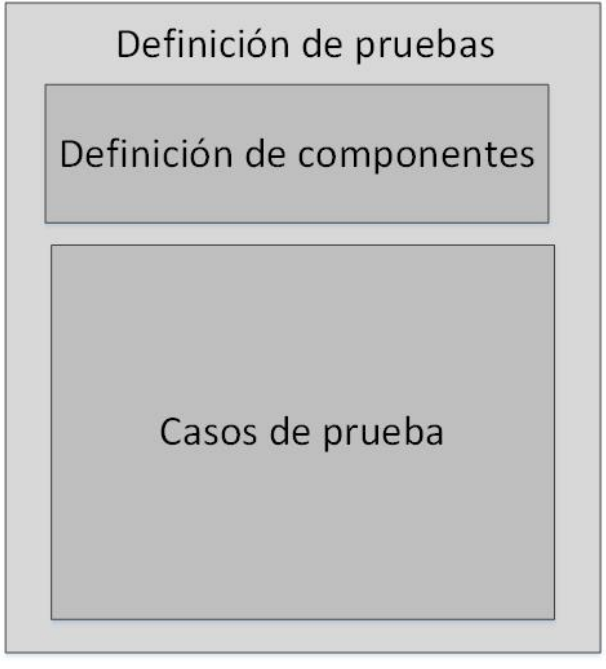
Definición de casos de prueba

Un caso de prueba es la agrupación o conjunto de pruebas (test) que el desarrollador de pruebas específica.

Un test es una expresión de prueba que puede tomar los siguientes valores:

- passed
- failed
- exception
- inconclusive

Para construir los veredictos de conjuntos de expresiones de prueba nos basamos en una lógica multivaluada, y definimos las tablas de verdad para las operaciones AND OR NOT.



¿Como diseñar una prueba?

- Escritura de pruebas en archivo PaSCAni
- Compilación y generación de proyecto SCA (automático)
- Ejecución de las pruebas sobre el sistema
- Visualización de los resultados en PaSCAni Explorer

A variability model for generating SCA-based matrix-chain multiplication software products

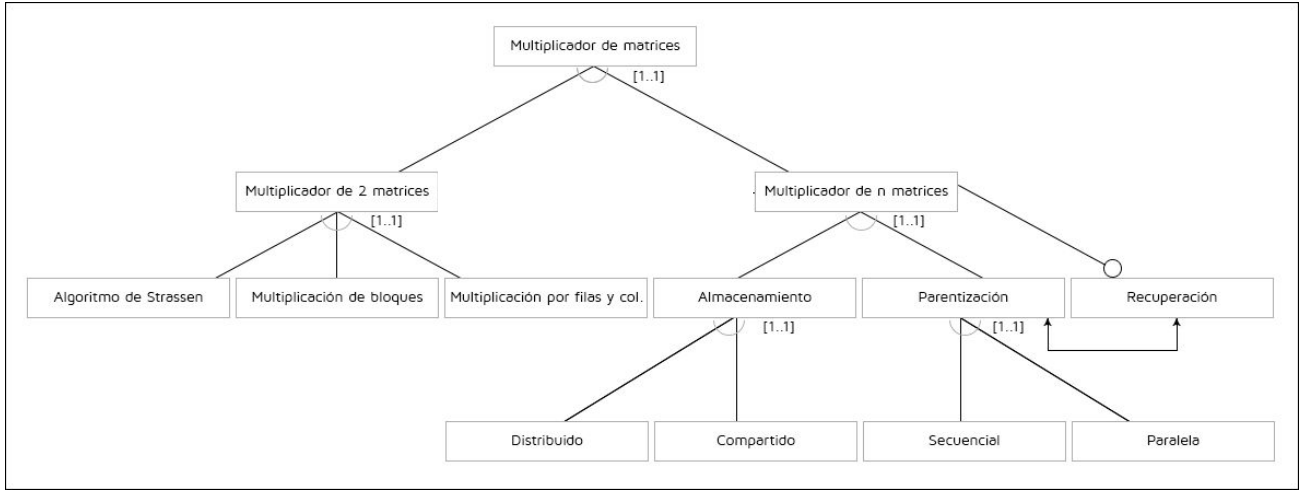
Propuesta de componentes SCA para hacer óptima la multiplicación de matrices, usando algoritmos ya conocidos, como el algoritmo de Volker Strassen, la aplicación de la parentización secuencial y paralela, entre otros. Usando diferentes estrategias de distribución adaptadas a SCA y tomadas de una propuesta basada en la arquitectura map/reduce.

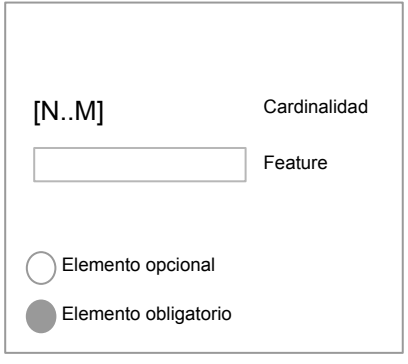
Desarrollo del caso de estudio

- Modelo de variabilidad
- Configuración de productos
- Módulos de prueba PaSCAni
- Análisis sobre la efectividad de las pruebas

Desarrollo del caso de estudio

- Modelo de variabilidad
- Configuración de productos
- Módulos de prueba PaSCAni
- Análisis sobre la efectividad de las pruebas





Configuración de productos

1. Strassen
2. Multiplicación de bloques (submatrices)
3. Multiplicación de filas y columnas de bloques
4. Multiplicación de n matrices usando (2)
5. Multiplicación de n matrices usando (3)

Módulos de prueba PaSCAni: Strassen (1)

```
package org.driso.matrices.strassen.resources;
java-import org.driso.matrices.common.*;
java-import java.util.Arrays;

module StrassenTest {

    composite strassen = loadComposite("Strassen.composite") providing services {
        rmi multiplication@"localhost":1099 with interface
        org.driso.matrices.strassen.interfaces.MatricesMultiplicationService
    } using libraries {
        "strassen/mcm-strassen.jar",
        "lib/mcm-common.jar"
    };

    testsuite checkCorrectness(int[][] A, int[][] B) using strassen.multiplication {

        . . .

    }


    testsuite compareWithClassicStrassen(int[][] A, int[][] B, long classicStrassenTime)
    using strassen.multiplication {
        . . .
    }

    control {
        . . .
    }
}
```

Módulos de prueba PaSCAni: Strassen (2)

```
testsuite checkCorrectness(int[][] A, int[][] B) using strassen.multiplication {

    int[][] C = strassen.multiplication.multiply(A, B);
    int[][] naiveC = MatrixUtils.multiplyUsingNaiveAlgorithm(A, B);

    Console.log("Correctness ok(?): " + Arrays.deepEquals(C, naiveC));

    test correctness = Arrays.deepEquals(C, naiveC)
        labeled "Multiplication's correctness"
        message when passed: "The algorithm is well implemented"
                    failed: "Wrong implementation"
                    exception: "An exception has been thrown"
                    inconclusive: "There is no conclusion";

    return correctness;
}
```

Módulos de prueba PaSCAni: Strassen (3)

```
testsuite compareWithClassicStrassen(
    int[][] A,
    int[][] B,
    long classicStrassenTime
) using strassen.multiplication {

  long measured = exectime strassen.multiplication.multiplicate(A, B);
  Console.log("Strassen SCA measured time: " + measured);

  test isUnderExpectedTime = measured <= classicStrassenTime
      labeled "Comparisson between Strassen SCA and Strassen POJO"
      message when passed: "Strassen SCA's execution time is less or" +
                           "equal than Strassen POJO's execution time"
                failed: "It does not worth executing Strassen SCA"
                exception: "An exception has been thrown"
                inconclusive: "There is no conclusion";

    return isUnderExpectedTime;
}
```

Módulos de prueba PaSCAni: Strassen (4)

```
control {

    String pathA = "/home/user/app/matrices/images/A1";
    String pathB = "/home/user/app/matrices/images/A2";

    int[][] A = ImageUtils.getImageData(ImageUtils.readImageFromLocalURL(pathA));

    int[][] B = ImageUtils.getImageData(ImageUtils.readImageFromLocalURL(pathB));

    MatrixUtils mutils = new MatrixUtils();
    long classicStrassenTime = exectime mutils.multiplyUsingStrassen(A, B);

    compareWithClassicStrassen(A, B, classicStrassenTime);
    checkCorrectness(A, B);

}
```

Análisis

- El proceso de escritura de los módulos de prueba es relativamente rápido y fácil.
- La integración que realiza PaSCAni entre instrucciones Java e instrucciones propias facilita al desarrollador de pruebas la escritura de los casos de prueba, usando sus propias librerías y sin preocupaciones referentes al consumo y la provisión de servicios.
- Adicionalmente, el compilador de PaSCAni hace transparente la complejidad de la implementación del código en Java necesario para ejecutar las pruebas a nivel de servicio
- El total de líneas de código escritas en PaSCAni para el caso de estudio fue de **306**. Y el total de líneas java en el proyecto generado fue de **6280 + 684** de XML (6964 en total).

```
strassen
 |--src
    |--org
       |--pascani
          |--commons
             |--templates
                |--TestsResultsTemplate.soy
                --TestsResultsTemplateSoyInfo.java
             |--Commons.java
             --Console.java
             --Service.java
             --Test.java
             --TestsResultsGenerator.java
          |--components
             |--RunnerImpl.java
             --StrassenTest_CheckCorrectnessImpl.java
             --StrassenTest_ControlImpl.java
             --StrassenTest_CompareWithClassicStrassenImpl.java
          |--resources
             |--runner.composite
             --StrassenTest.composite
          |--services
             |--RunnableControl.java
             --RunnableTestsuite.java
 |--lib
   |--soy-latest.jar
 |--StrassenTest.jar
 --pascani-configuration.properties
 --testresults.xml
```

Contenido

Aspectos metodológicos

Etapas del proyecto:

- Investigación

  SAS, ULSS, SCA, Compiladores y lenguajes, entre otros.

- Propuesta de la solución

  Diseño de la gramática y propuesta de modelo de traducción y de ejecución

- Implementación

- Pruebas

  Caso de estudio: *A variability model for generating SCA-based matrix-chain multiplication software products*

Contenido

Entregables

- Implementación de PaSCAni (Compilador y Modelo de ejecución) en gForge
- Implementación de PaSCAni explorer
- Documento de proyecto de grado
- Configuración de productos de software para el caso de estudio

Objetivos específicos

- Proponer un conjunto de reglas léxicas, sintácticas y semánticas que permitan especificar requerimientos funcionales para generar y ejecutar clases de prueba, en tiempo de ejecución.

- Implementar un prototipo funcional de un componente de software que permita realizar validación y verificación de un sistema auto-adaptativo en tiempo de ejecución, empleando un conjunto de especificaciones usando las reglas léxicas, sintácticas y semánticas del lenguaje propuesto.

- Integrar el prototipo funcional propuesto al caso de estudio *A variability model for generating SCA-based matrix-chain multiplication software products*.

Resultados y Conclusiones (1)

- Desarrollamos una nueva forma de validar y verificar sistemas de software basados en componentes SCA.
- PaSCAni reduce altamente el esfuerzo del desarrollador de pruebas al escribir y ejecutar pruebas sobre el sistema.
- Estamos proponiendo una solución versátil, útil no sólo en el contexto de los sistemas de software basados en componentes sino también en los sistemas auto-adaptativos, como una línea base para, gradualmente, construir funcionalidades más grandes y complejas, como la auto-recuperación.

Resultados y Conclusiones (2)

- A problemas específicos, soluciones específicas. Los DSL hacen transparente la complejidad de las tecnologías y **permiten al desarrollador enfocarse en la solución**.

- Es necesario enfrentar la carencia de semántica en las soluciones de software. El usuario debe sentir que **resuelve un problema a partir de conceptos conocidos y afines al contexto correspondiente**.

- Un **enfoque modular en los mecanismos de validación y verificación**, en tiempo de ejecución, contribuye no sólo a la realización de pruebas integrales sino también a garantizar el cumplimiento de los requerimientos en sistemas cuya arquitectura cambia constantemente.

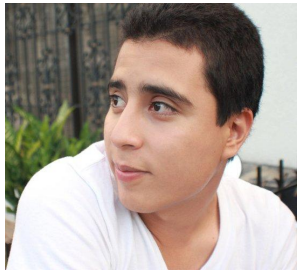
- La ejecución de las pruebas preliminares de PaSCAni son aceptables en tiempo de respuesta: compilación xms y esto es muy comparable a los resultados de compilación de java e igualmente los de ejecución.

Contenido

Trabajo futuro

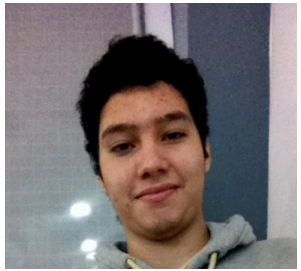
- Mejorar la gramática de PaSCAni, en términos de expresividad.
- Desarrollar herramientas que le permitan al desarrollador de pruebas ser más productivo (editor, plugin de eclipse para PaSCAni Explorer).
- Integrar PaSCAni a un sistema de software auto-adaptativo y desarrollar las clases necesarias para realizar auto-recuperación basada en la ejecución de pruebas propuesta.

Preguntas



Fabián Andrés Caicedo Cuellar

fabian.caicedo@correo.icesi.edu.co



Miguel Ángel Jiménez Achinte

miguel.jimenez@correo.icesi.edu.co

UNIVERSIDAD ICESI